客户端应用架构

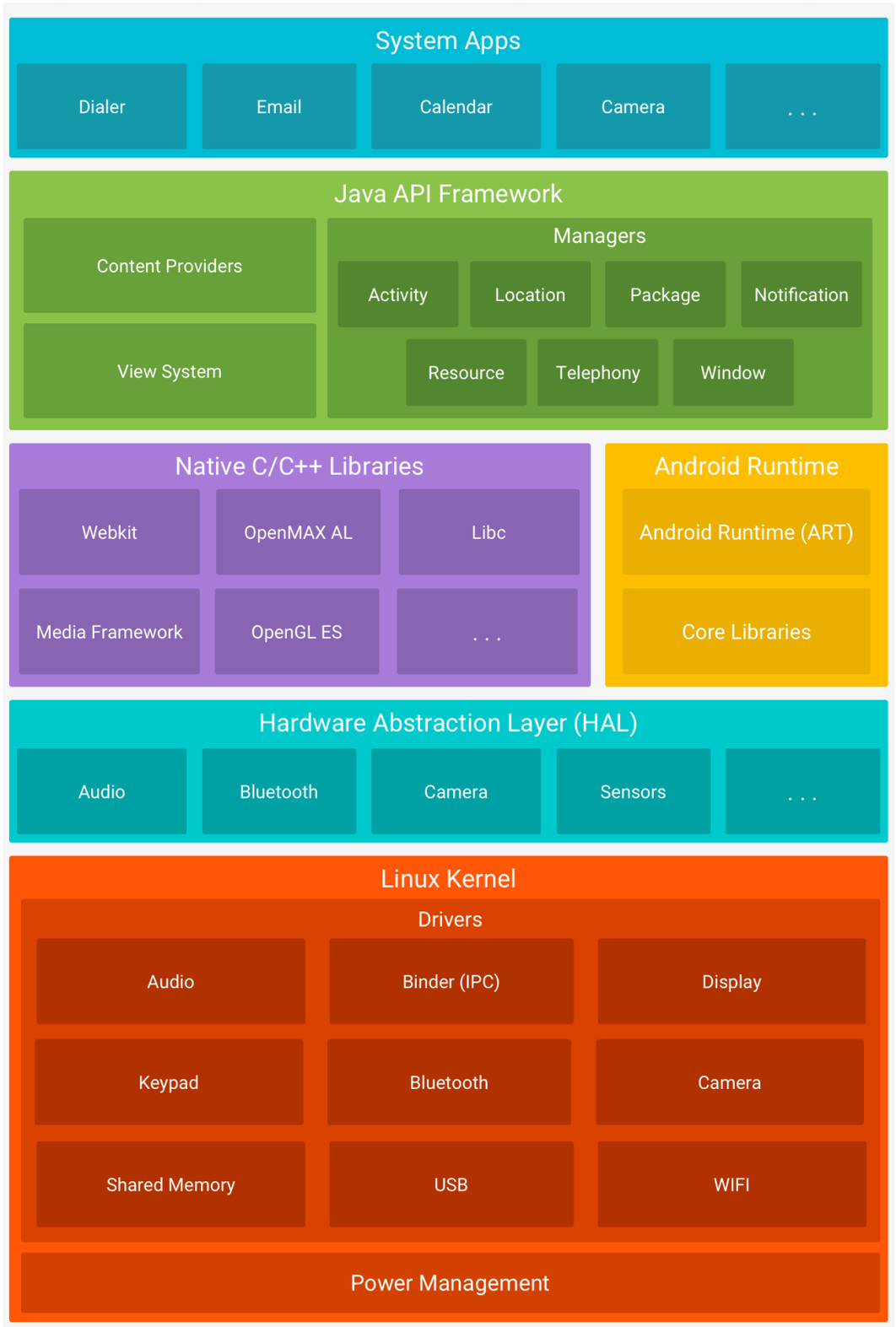
085 陈峰

要讲什么

- 什么是架构
- App架构三剑客（MVC -> MVP -> MVVM）
- 如何选择合适的架构

什么架构?

Android Architecture



Mac OSX

Cocoa Layered Architecture - Mac OSX

www.knowstack.com
By: Debasis Das

Cocoa Application - Application User Interface Responds to User Events, Manages App Behavior

App Kit | Notification Center | Game Center | Sharing | Full Screen Mode | Cocoa Autolayout | Popovers | Software Configuration | Accessibility | Apple Script | Spotlight

Media Plays , records, editing audiovisual media, Rendering 2D and 3D graphics

- **AV Foundation**
  - Audio Playback, editing, Analysis & Recording
- **Core Animation**
  - 2D rendering & Animation
  - 3D Transformations
- **Core Audio**
  - Audio Services for recording, playback and synchronization
- **Core Image**
  - Fast Image Processing
  - Uses GPU Based acceleration
- **Core Text**
  - Handles Unicode Fonts & texts
- **Open AL**
  - Delivers 3D Audio
  - High performance positional playbacks in games
- **Open GL**
  - Portable 3D graphics apps & Games
  - Imaging functions & Effects
- **Quartz**
  - OSX Graphics, Rendering support for 2D content
  - Event Routing & Cursor Management

Core Services - Fundamental Services for low level network communication, Automatic Reference Counting, Data Formatting, String Manipulation

- **Address Book**
  - Centralized Database for contacts & groups
- **Core Foundation**
  - declares C based programmatic interfaces
  - Data Types & Data Management
- **Quick Look**
  - Enables Spotlight & finder to display thumbnail images
- **Security**
  - User Authentication, Certificates & keys, Authorization, Keychain Services etc
- **Core Data**
  - Data Model Management & Storage, Undo/Redo, Validation of property values
- **Foundation**
  - Objective C Framework for Object Behavior, Internationalization, Data Types & Data Management
- **Social**
  - Supports integration with Social Networking services
- **Webkit**
  - Display HTML Content in apps. contains WebCore and JavaScript Core

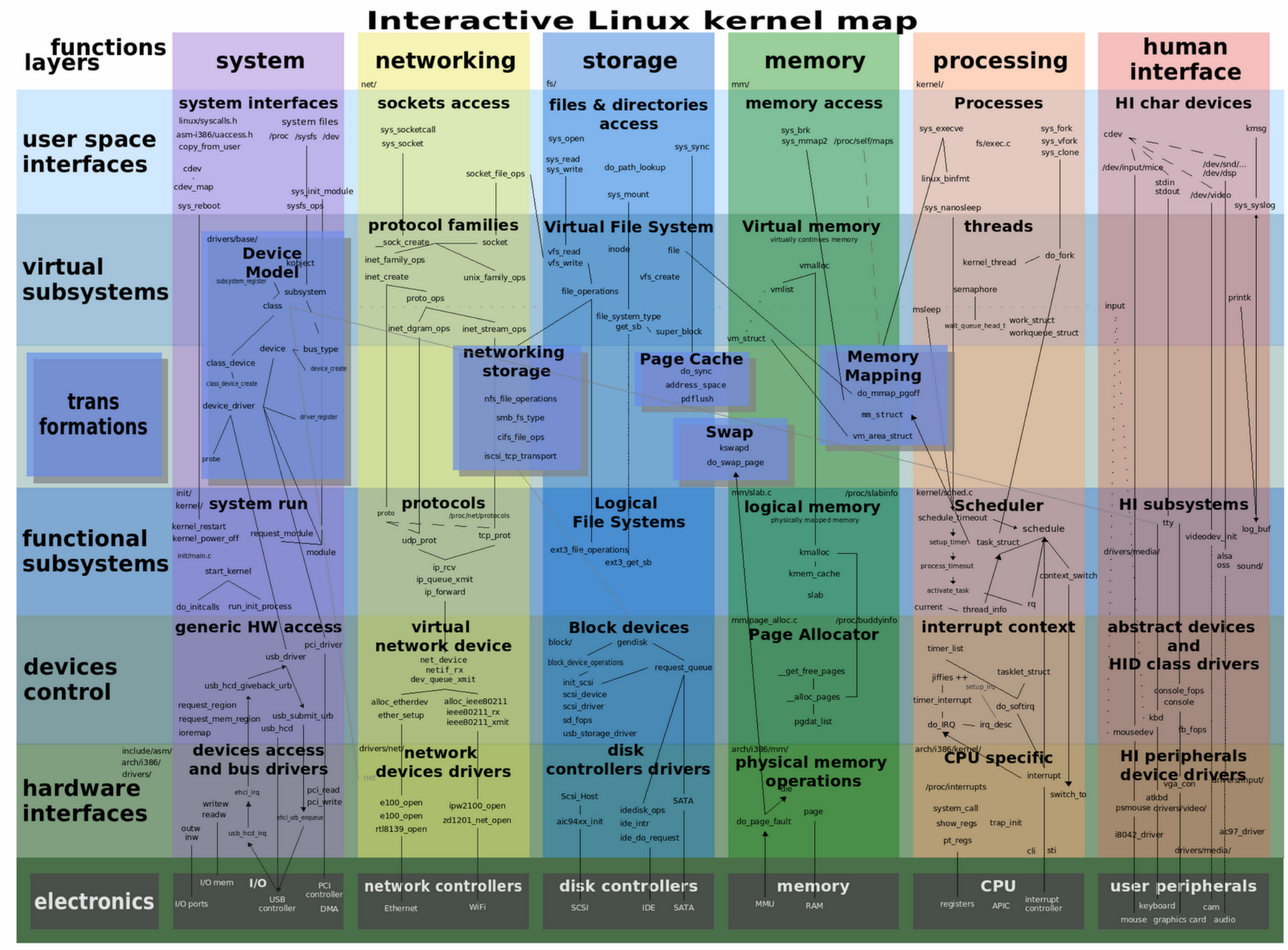
Core OS - Related to hardware and networking. Interfaces for running high-performance computation tasks on CPU or GPU

- **Accelerate**
  - Accelerate complex operations, improve performance using vector unit, Supports data parallelism, 3d Graphic imaging, image processing
- **Directory Services**
  - Provides access to collected information about users, groups, computers, printers in a networked environment
- **Disk Arbitration**
  - Notifies when local or remote volumes are mounted and unmounted
- **Open CL**
  - Makes the high-performance parallel processing power of GPUs available to general purpose computing
- **System Configuration**
  - Provides access to current network configuration information. Determines reachability of remote hosts. Notifies about change in network

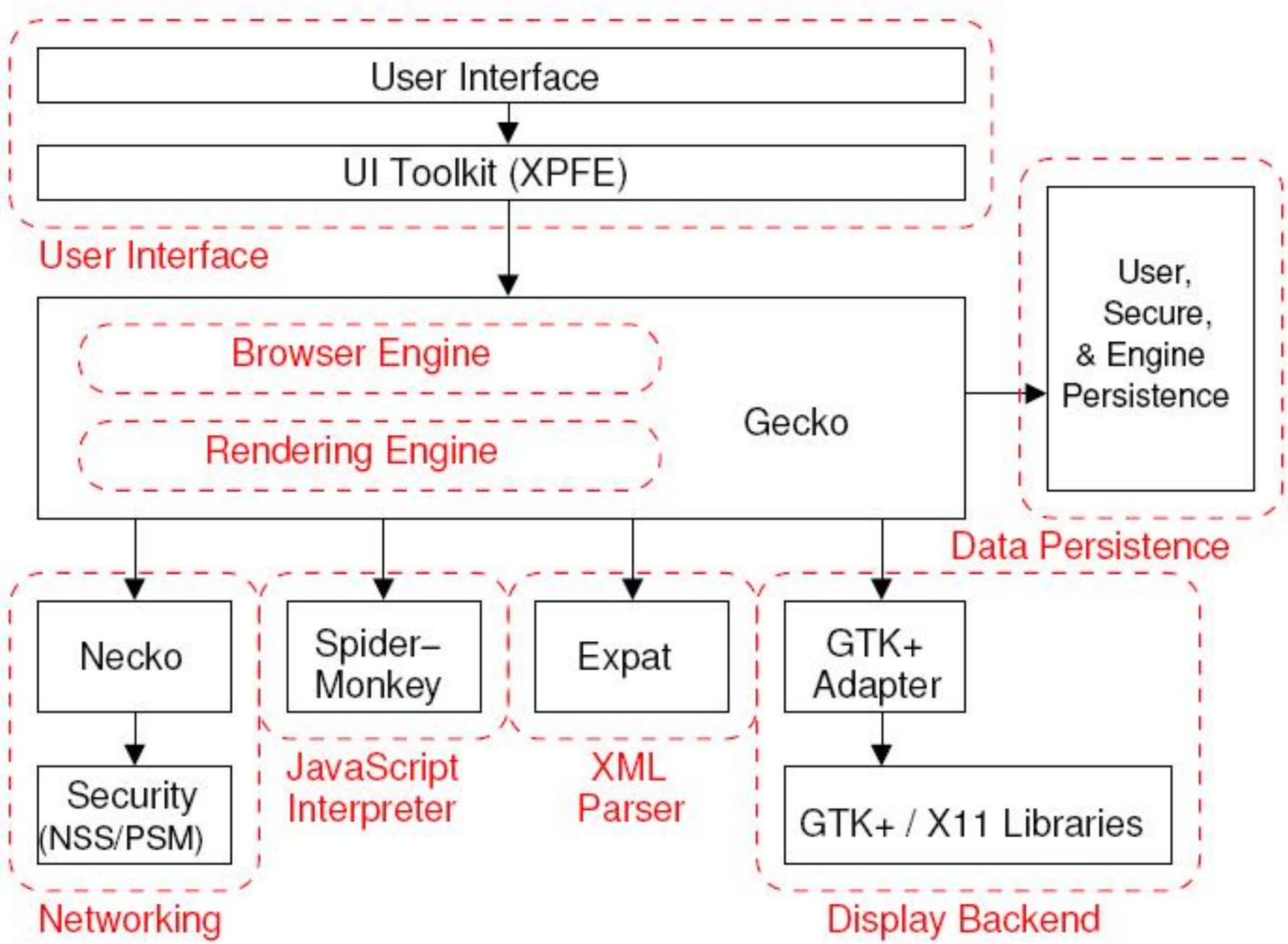
Kernel & Device Drivers - Device drivers & BSD Libraries , low level components. Support for file system security, interprocess communications, device drivers

- **BSD**
  - Provides basis for file systems and networking facilities, POSIX Thread support, BSD Sockets
- **File System**
  - Supports multiple volume formats (NTFC, ExFAT, FAT etc) & File Protocols (AFP, NFS etc)
- **Mach**
  - Protected Memory, Preemptive multitasking, Advanced Virtual Memory, Real Time Support
- **Networking**
  - Supports network kernel extensions (NKEs). Create network modules, Configure protocol stacks, Monitor and modify network traffic

Linux Kernel

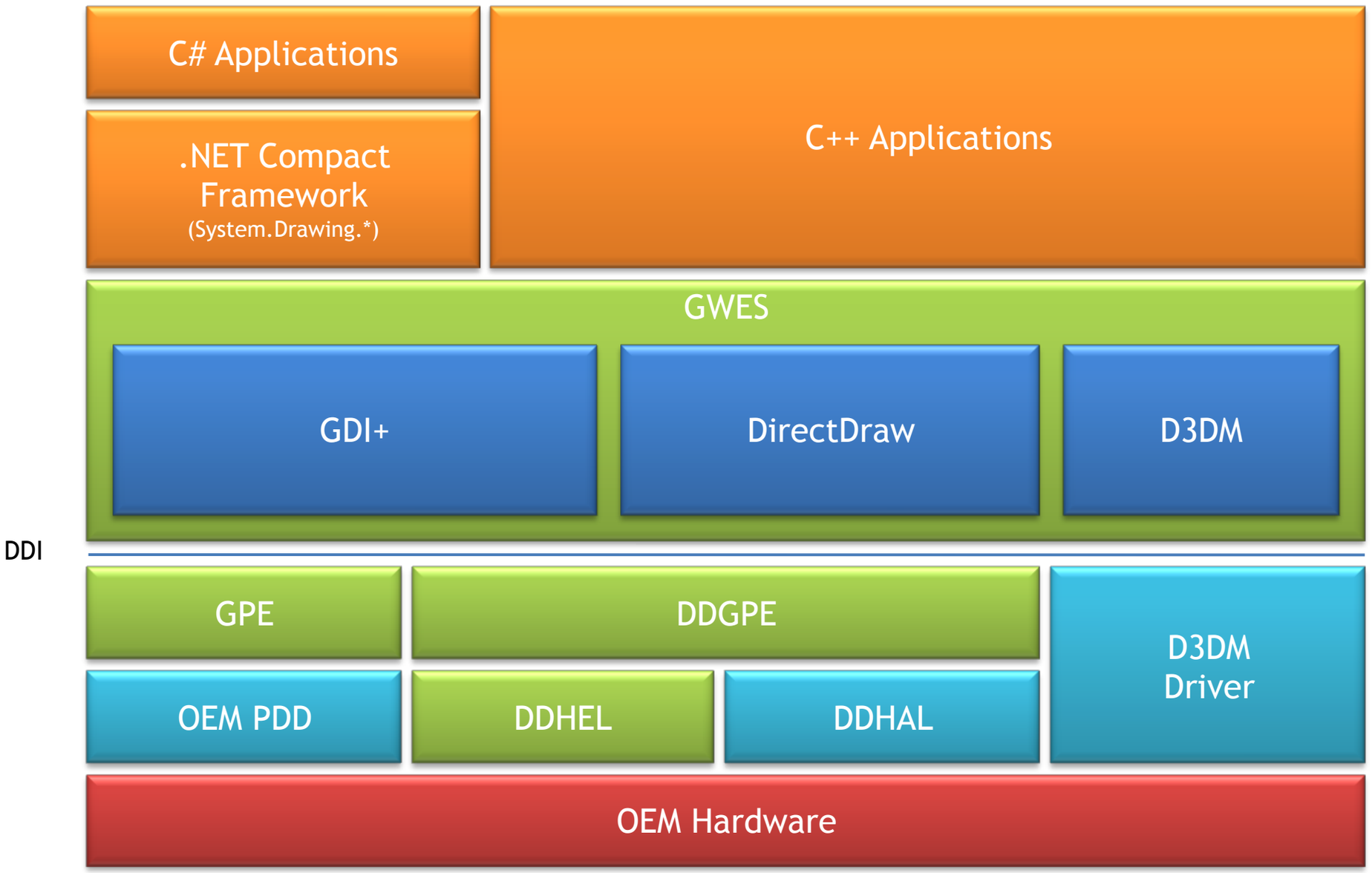


Mozilla Firefox



Windos Mobile

GWES: Graphics, Windowing, and Event Subsystem
GPE: Graphics Primitive Engine
D3DM: Direct3D Mobile
DDI: Display Device driver interface

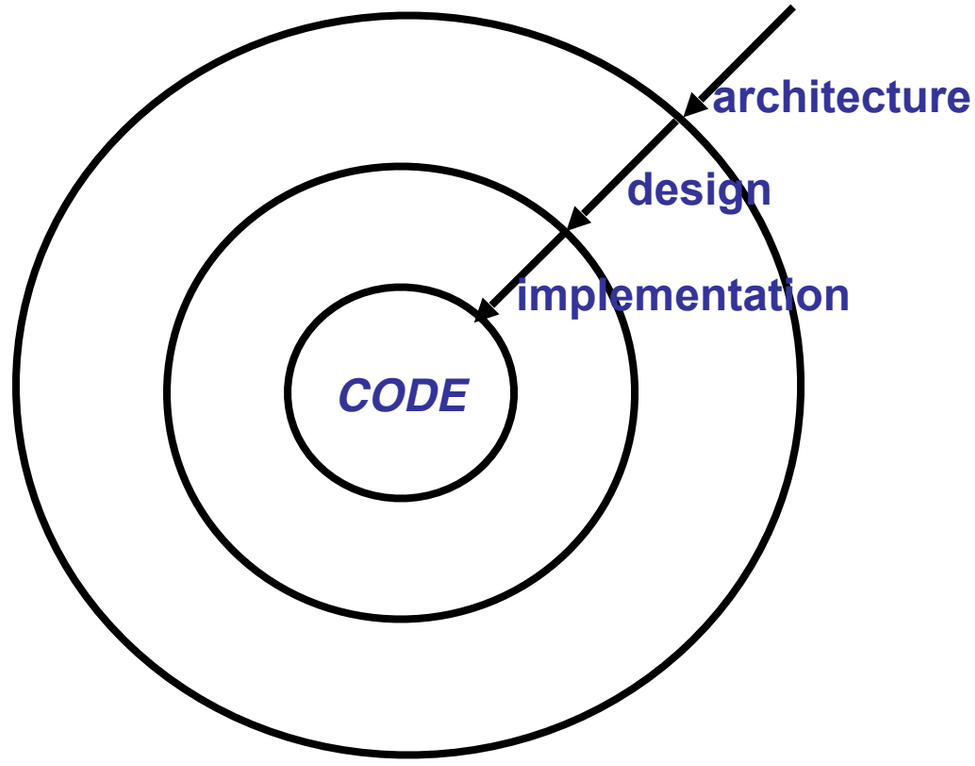


软件架构定义

- 没有一个统一的定义
  - http://www.sei.cmu.edu/architecture/start/community.cfm
- 比较好的一个定义
  - Software architecture encompasses the set of significant decisions about the organization of a software system

Architecture

- All architecture is design, but not all design is architecture
- Architecture focuses on significant design decisions, decisions that are both structurally and behaviorally important as well as those that have a lasting impact on the performance, reliability, cost, and resilience of the system
- Architecture involves the how and the why, not just the what



糟糕架构带来的问题

- 不易分工：
  - 多人共同开发一个复杂的App，由于没有清晰的架构，导致分工不明确，职责不清，工期变长。
- 可读性差：
  - 刚接手一个新的App，但是发现所有功能都混杂在一起，要增加一个小功能，必须要把所有逻辑都看懂。
- 不易测试：
  - 所有功能都混杂在一起，耦合性太强，无法进行独立测试。
- 难以维护：
  - 几乎所有的主要逻辑都集中在几个文件中，每个文件都有几千行代码，功能复杂，难以扩展和维护。

优秀架构的特点

- 满足产品需求
- 层级和模块清晰，易于分工
- 性能足够好
- 功能稳定、可靠、安全
- 代码易读，容易修改和测试
- 容易扩展和维护
- ……

问题讨论

- 下面这3个概念有什么区别?
  - 设计模式 (Design)
  - 程序框架 (Framework)
  - 软件架构 (Architecture)

- 设计模式：是解决开发中遇到问题的一种解决方案，是一些最佳实践，例如Gof 23种设计模式
- 框架：是指面向特定需求的、可复用的“半成品”软件。例如依赖注入框架，网络框架，ORM框架，自定义View框架，动画框架等
- 软件架构：不是软件，是关于软件如何设计的重要决策，通常会形成一个系统的、全局的草图，其中会包含一些设计模式和框架

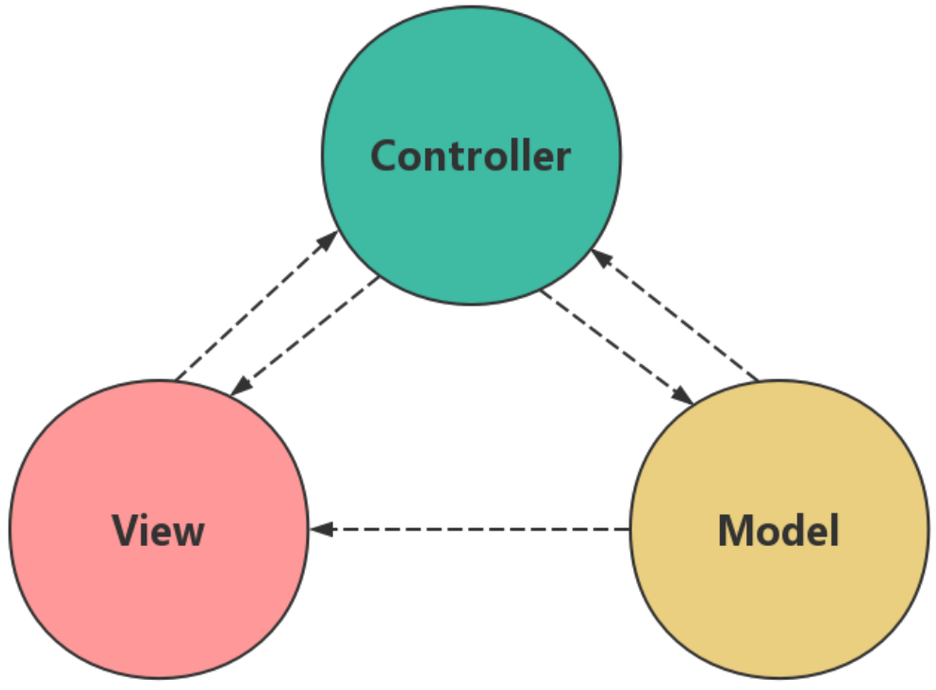
MVC -> MVP -> MVVM

MVC

- 不同平台上的MVC架构
  - Spring MVC
  - Rails MVC
  - Python Django MVC (MTV)
  - iOS MVC
  - Android MVC

MVC

- M（模型层）：Model一般用来保持程序的数据状态，比如数据存储、网络请求等。
- V（视图层）：View一般由GUI组件组成，同时响应用户的交互行为并触发Controller的逻辑。比较好的View设计应该是被动的，只负责向用户展示以及交互。
- C（控制器）：Controller由View根据用户行为触发并响应来自View的用户交互，然后根据View的事件逻辑修改对应的Model，Controller并不关心View如何展示相关数据或状态，而是通过修改Model并由Model的事件机制来触发View的刷新。



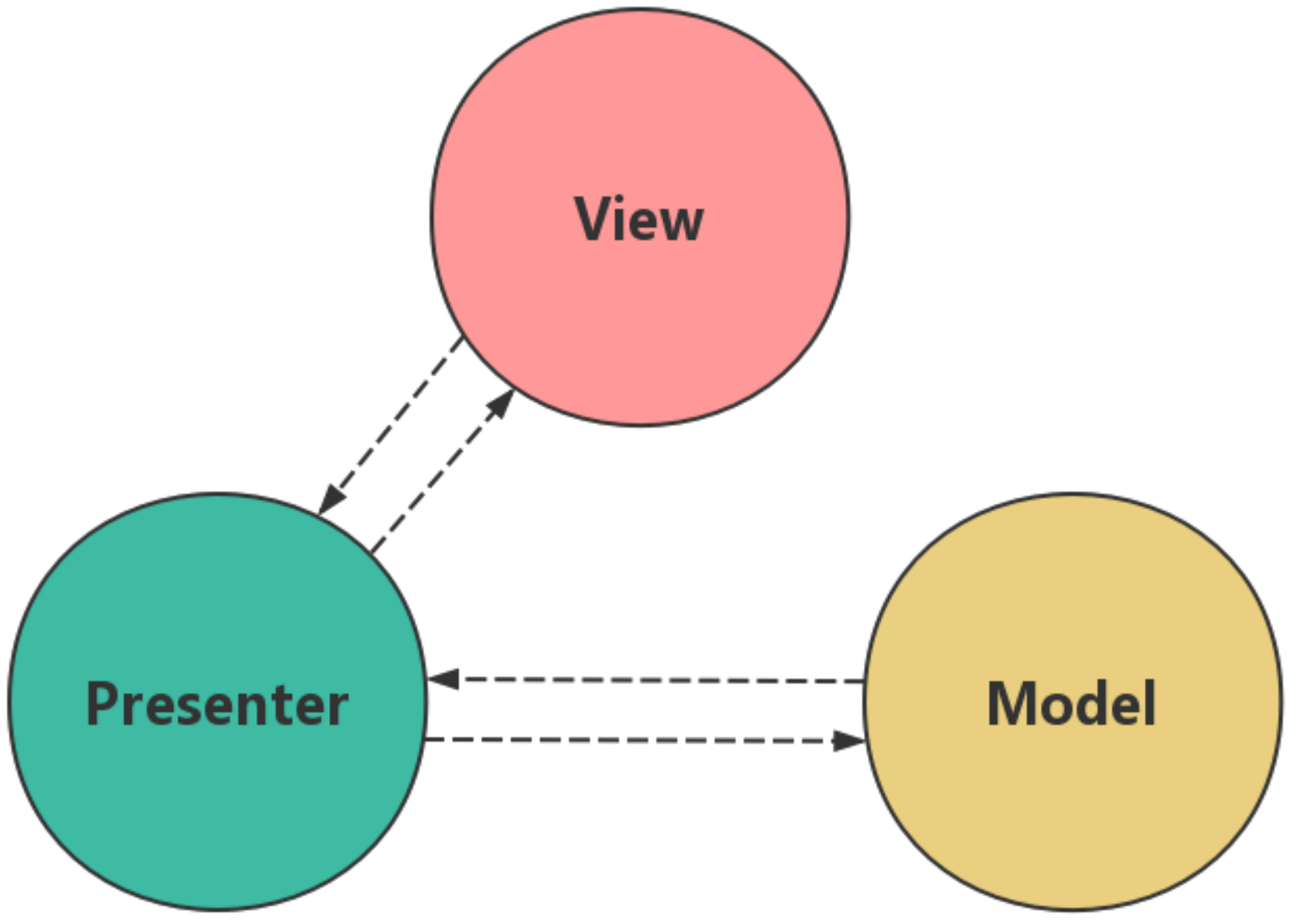
MVC in Android

- C的职责：
  - 管理UI以及各个UI视图的生命周期
  - 展示内容和布局
  - 处理用户行为 (如按钮的点击和手势的触发等)
  - 储存当前界面的状态 (例如当前页面状态、是否正在进行网络请求等)
  - 处理界面的跳转
  - 业务逻辑和各种动画效果
  - 各种请求逻辑（网络、数据库）
  - 数据处理逻辑（排序、分类）
  - …
- 导致Activity/Fragment/View中动辄上千行代码

MVC不足

- Controller很容易变得臃肿
  - Android中因为业务逻辑和数据存取是紧耦合的，开发人员很可能会吧各种各样的业务逻辑塞进Activity、Fragment或者自定义View中
  - Controller中又可能包含很多的异步任务，导致复杂
- View和Model存在耦合性，应用很难进行分层
- 导致难以维护，难以扩展

MVP设计图



MVP

- Presenter（交互中间人）
  - Presenter主要作为沟通View和Model的桥梁，它从Model层检索数据后，返回给View层，使得View和Model之间没有耦合，也将业务逻辑从View角色上抽离出来。
- View（用户界面）
  - 通常指Activity、Fragment或者自定义View，它含有一个Presenter成员变量。通常View需要实现一个逻辑接口，将View上的操作转交给Presenter进行实现，最后Presenter调用View逻辑接口将结果返回给View元素。
- Model（数据的存取）
  - 主要提供数据的存取功能。Presenter通过Model层存储、获取数据，Model就像一个数据仓库。（例如：Model封装了数据库DAO或者网络获取数据功能）

MVP

- 相比MVC
  - 能够有效的降低View的复杂性，避免业务逻辑被塞进View中
  - 能够解除View与Model的耦合，带来了良好的可扩展性、可测试性
  - 易于扩展，M-V-P三者之间的关系是通过接口建立的，是一种松耦合关系。这样当UI发生变化，我们只需要替换View即可；而数据库引擎需要替换时，只需要重新构建一个实现了Model相关接口的类即可。

MVP不足

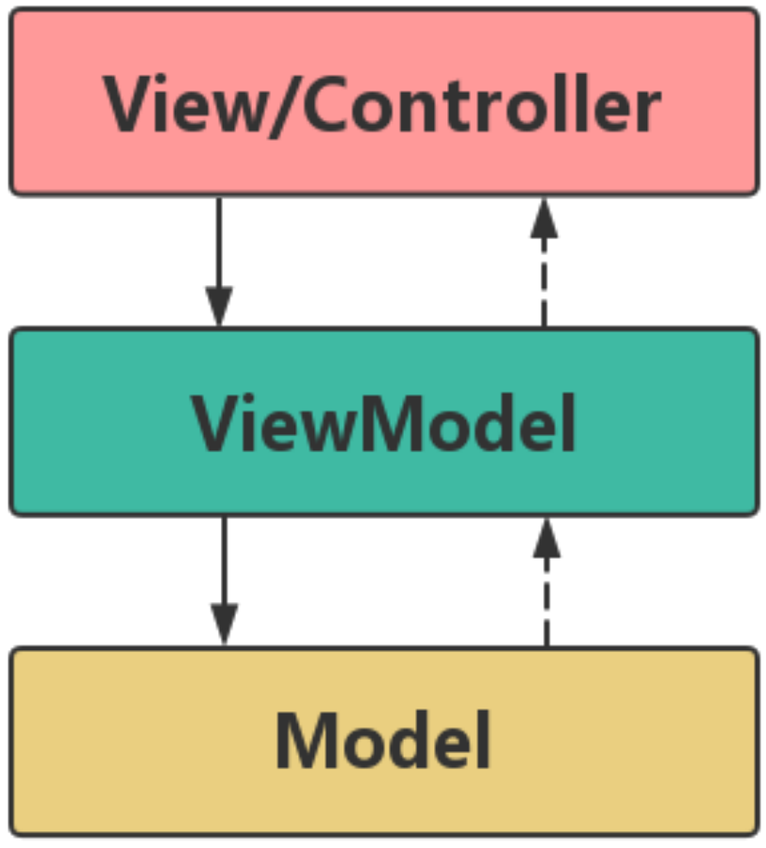
- 几点不足
  - Presenter层要处理的业务逻辑过多，复杂的业务逻辑会使P层非常庞大和臃肿
  - Presenter是通过抽象接口实现Model和View解耦的，随着View和Model越来越复杂，需要增加的抽象接口会非常多，导致接口文件会非常多（实际开发中，抽象的粒度很难统一、很难控制）
  - Presenter和View的生命周期并不一致，在Presenter中需要额外监听View的生命周期
- 不过，MVP确实是一个不错的分层应用架构，实际应用也非常多

MVVM

- MVVM有哪些实现
  - WPF (Windows Presentation Foundation)
  - VUE
  - **Android MVVM**
  - iOS MVVM
  - Unity MVVM

MVVM

- MVVM
  - 将View层和Controller层进行了合并，统称为View层
  - 引入了一个新的模块 — ViewModel层，主要承载View展现逻辑
  - Model层和MVC、MVP没有本质区别
- 可以看做是MVP的改进版本，也是一种分层架构
  - Controller更加清晰简洁
  - ViewModel分离了逻辑，方便测试
  - 开发解耦



MVVM - DataBinding

- MVVM基本上和MVP模式一致，主要区别在于MVVM采用了数据双向绑定
  - 通过将View层和ViewModel层进行双向绑定， View层的变化会自动通知给ViewModel层，而ViewModel层的数据变化也会通知给View层进行相应的UI的更新。
- 双向数据绑定
  - 在ViewModel里面有一个Binder或者DataBinding Engine的东西，负责View和Model之间的数据操作

DataBinding Sample

- 现在用户下拉刷新一个页面，页面上出现10条新的新闻，新闻总数从10条变成20条。那么MVC、MVP、MVVM的处理依次是：

  1. View获取下拉事件，通知Controller
  2. Controller向后台Model发起请求，请求内容为下拉刷新
  3. Model获得10条新闻数据，传递给Controller
  4. Controller拿到10条新闻数据，可能做一些数据处理，然后拿处理好的数据渲染View

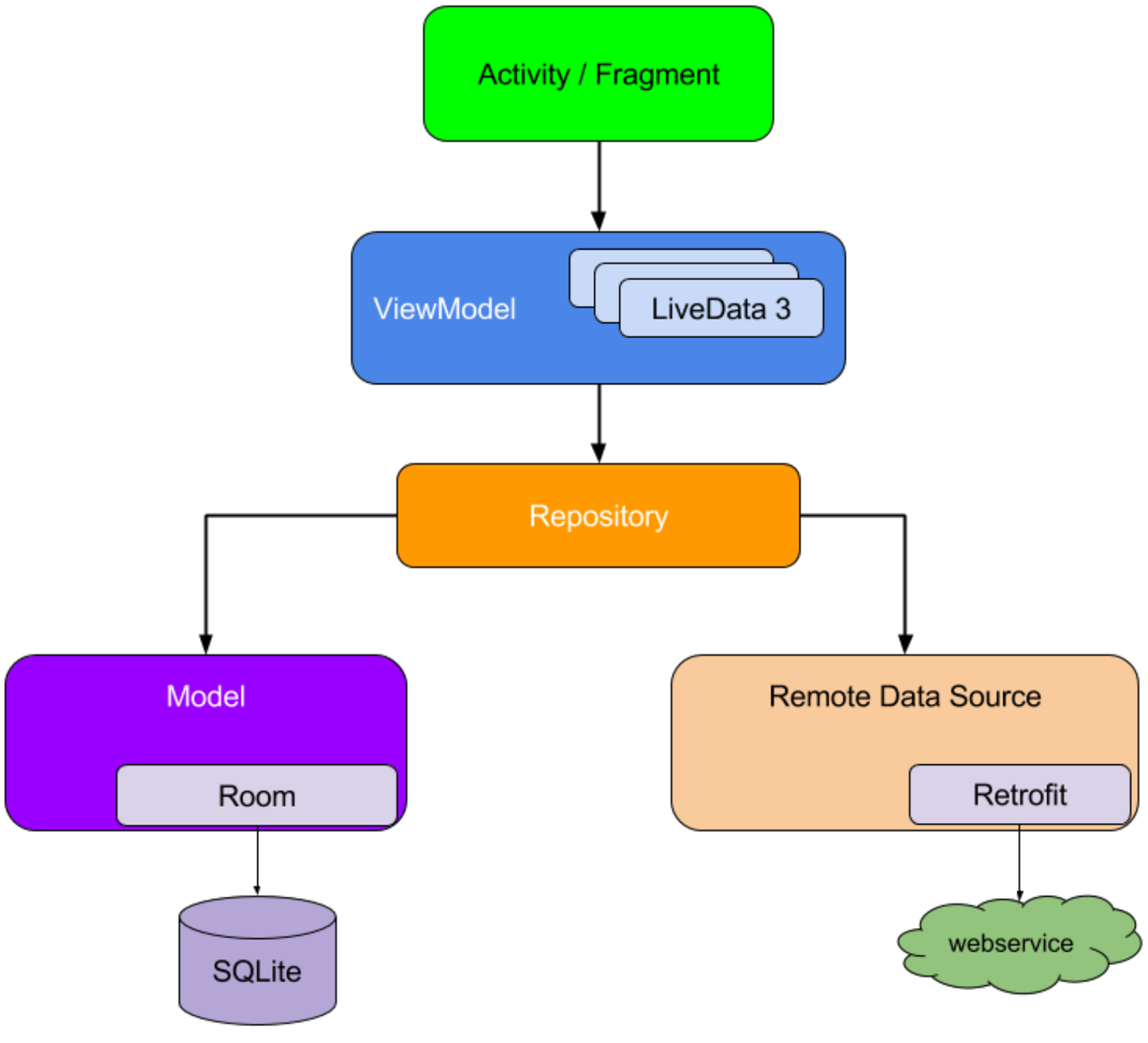
  - MVC: Controller拿到UI节点，渲染10条新闻
  - MVP: Presenter通过View提供的接口渲染10条新闻
  - MVVM: 无需操作，只要VM的数据变化，通过数据双向绑定，View直接变化

DataBinding in Android

- Google提供了一种解决方案，只需要在gradle文件中添加如下代码

```
android {

    dataBinding {
        enabled = true
    }

}
```

MVVM in Android



LiveData

- LiveData是一个数据持有类，它可以通过添加观察者被其他组件观察其变更。不同于普通的观察者，它最重要的特性就是遵从被观察者的生命周期，如在Activity中如果数据更新了但Activity已经是destroy状态，LiveData就不会通知Activity(observer)。
- 工作原理
  - https://juejin.im/post/5baee5205188255c930dea8a

LiveData (1/2)

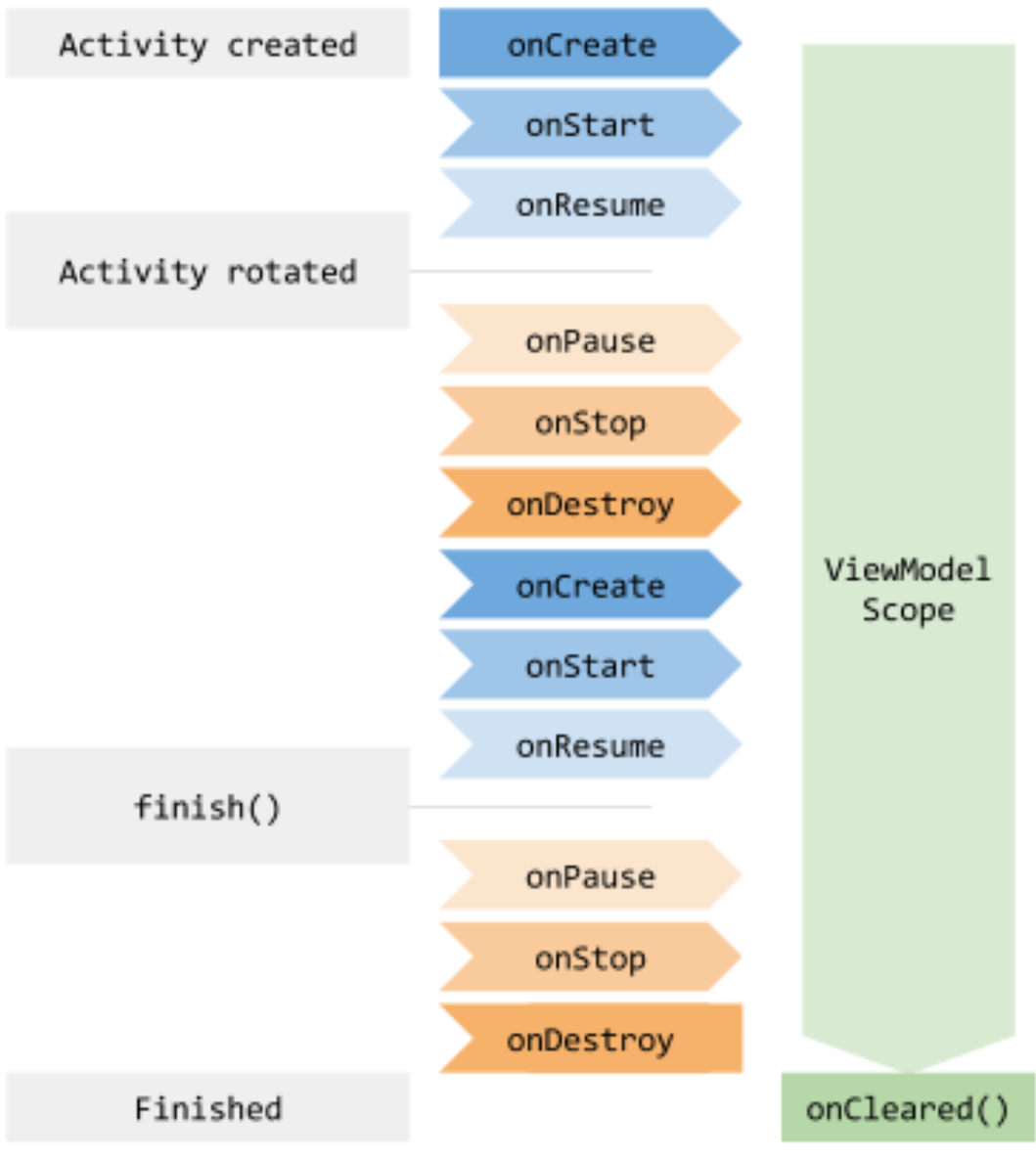
- LIveData确保你的UI使用的是最新的数据，而且还不需要你在数据变化时自己调用更新。
  - 不会内存泄漏：因为Observer只会在自己的生命周期拥有，LifeCycleOwner active的状态下，才会接收到数据更新的通知，所以不会再有内存泄漏的问题
  - 不会在Activity销毁后还接收到更新UI的指令导致崩溃。
  - 开发者不需要再手动管理生命周期。
  - 即使在配置发生变化时，比如activty的横竖屏切换，观察者还是能收到最新的有效数据。
- LiveData可以以单例模式在应用中存在。可以提供给任何需要它的观察者监听，以实现多页面多处监听更新

LiveData (2/2)

- 常用数据类型
  - MutableLiveData： 以public形式暴露了LiveData的setValue和postValue方法
  - MediatorLiveData：可以观察其他的LiveData数据，并在其发生变化时通知给MediatorLiveData

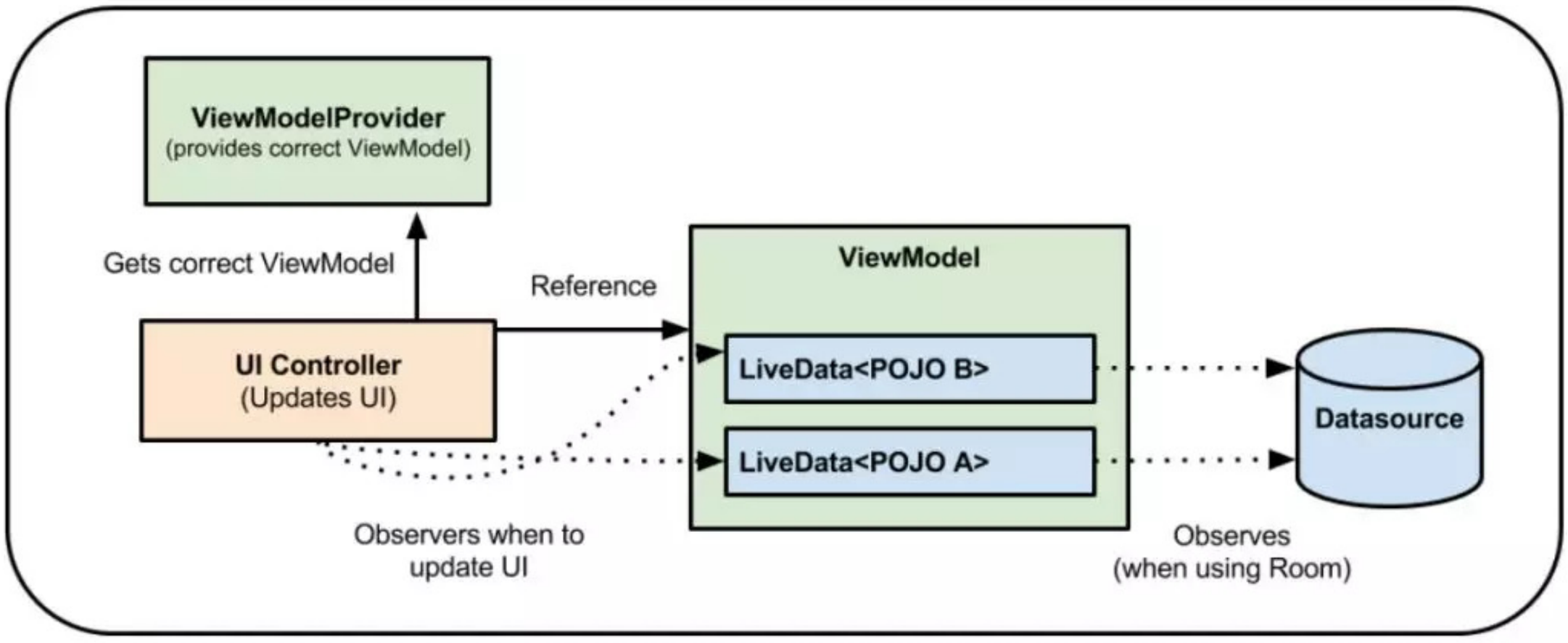
ViewModel

- ViewModel负责为View层准备数据
  - 如果 Activity 因为配置发生变化而被重建了，那么当重建的时候，ViewModel 仍然是之前的实例
- 方便Fragment之间共享数据
- 结合Room和LiveData，代替了CursorLoader方式来获取数据和监听数据变化



Room

- Room是Android官方提供的新的数据持久化方案组件
  - Room 是在 Sqlite 之上添加的一个抽象层, 以便实现更加强大的数据库访问，其可直接返回 LiveData， 用于监听数据返回。



Room

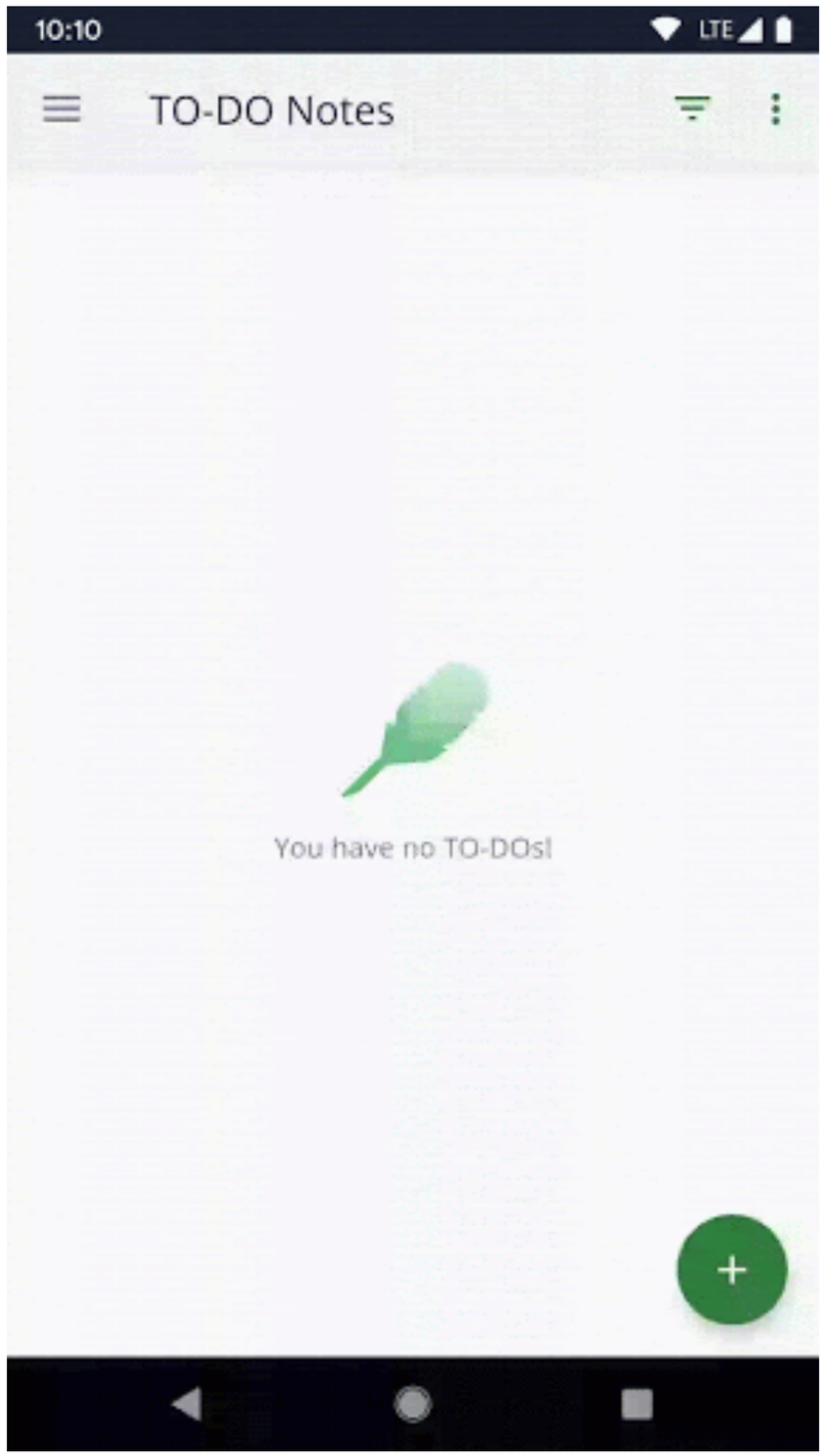
- Room中的几个重要注解
  - @Database: 注解继承自RoomDatabase的类，主要用于创建数据库和Daos(数据访问对象)。
  - @Entity :用来注释实体类，@Database类通过entities属性，引用被@Entity注解的类，并通过这个类的所有属性作为表的列名来创建数据库的表。
  - @Dao: 注解接口或抽象方法，用来提供访问数据库数据的方法。在使用@Database注解的类中必须定义一个不带参数的方法，这个方法返回使用@Dao注解的类。

Repository

- Android官方推荐我们使用Repository模式来进行数据获取逻辑的封装。
  - Repository可以给app其余模块提供一个干净的api接口，这样其他模块就会独立于数据获取的逻辑之外，能更专注于他们本身的业务范畴。
  - Repository知道从哪里获取数据，也知道数据更新时该如何调用。Repository可以将其他模块，比如本地持久化数据（Room），网络数据（Web Service），缓存数据（Cache）等串联起来。
- ViewModel只需要从Repository来更新数据
  - ViewModel并不知道，也不需要关心这些数据从哪里来以及怎么来。从而达到ViewMode是与数据获取逻辑独立开的目的。

Google Sample

- 推荐一个完整的官方Sample
  - https://github.com/googlesamples/android-architecture
- To-do App
  - 只有一个activity和一个fragment
  - 使用了LiveData和Data Binding来组织UI
  - 数据层使用了Room和Remote



To-do App: TaskActivity

- 在TasksActivity中，通过obtainViewModel方法获取TasksViewModel对象

```
public static TasksViewModel obtainViewModel(FragmentActivity activity) {
    // Use a Factory to inject dependencies into the ViewModel
    ViewModelFactory factory = ViewModelFactory.getInstance(activity.getApplication());

    TasksViewModel viewModel =
            ViewModelProviders.of(activity, factory).get(TasksViewModel.class);

    return viewModel;
}
```

To-do App: TaskActivity

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    ......

    mViewModel = obtainViewModel( activity: this);

    // Subscribe to "open task" event
    mViewModel.getOpenTaskEvent().observe(this, new Observer<Event<String>>() {
        @Override
        public void onChanged(Event<String> taskIdEvent) {
            String taskId = taskIdEvent.getContentIfNotHandled();
            if (taskId != null) {
                openTaskDetails(taskId);
            }

        }
    });

    // Subscribe to "new task" event
    mViewModel.getNewTaskEvent().observe(this, new Observer<Event<Object>>() {
        @Override
        public void onChanged(Event<Object> taskIdEvent) {
            if (taskIdEvent.getContentIfNotHandled() != null) {
                addNewTask();
            }
        }
    });
}
```

To-do App: TasksViewModel

- MutableLiveData类型

```
public class TasksViewModel extends ViewModel {

    ......

    private final MutableLiveData<Event<String>> mOpenTaskEvent = new MutableLiveData<>();

    private final MutableLiveData<Event<Object>> mNewTaskEvent = new MutableLiveData<>();
```

To-do App: TasksFragment

- 下面是TasksFragment的布局文件，其中绑定了TasksViewModel

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <import type="android.view.View" />

        <import type="androidx.core.content.ContextCompat" />

        <variable
            name="viewmodel"
            type="com.example.android.architecture.blueprints.todoapp.tasks.TasksViewModel" />

    </data>


    <androidx.coordinatorlayout.widget.CoordinatorLayout
        android:id="@+id/coordinator_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <com.example.android.architecture.blueprints.todoapp.ScrollChildSwipeRefreshLayout
            android:id="@+id/refresh_layout"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:onRefreshListener="@{viewmodel::refresh}"
            app:refreshing="@{viewmodel.dataLoading}">
```

To-do App: TasksFragment

```
<com.example.android.architecture.blueprints.todoapp.ScrollChildSwipeRefreshLayout
    android:id="@+id/refresh_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:onRefreshListener="@{viewmodel::refresh}"
    app:refreshing="@{viewmodel.dataLoading}">

    <RelativeLayout
        android:id="@+id/tasks_container_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:clickable="true"
        android:orientation="vertical">

        <LinearLayout
            android:id="@+id/tasks_linear_layout"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:orientation="vertical"
            android:visibility="@{viewmodel.empty ? View.GONE : View.VISIBLE}">

            ......

            <androidx.recyclerview.widget.RecyclerView
                android:id="@+id/tasks_list"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
                app:items="@{viewmodel.items}" />
        </LinearLayout>
        ......
    </RelativeLayout>

</com.example.android.architecture.blueprints.todoapp.ScrollChildSwipeRefreshLayout>
```

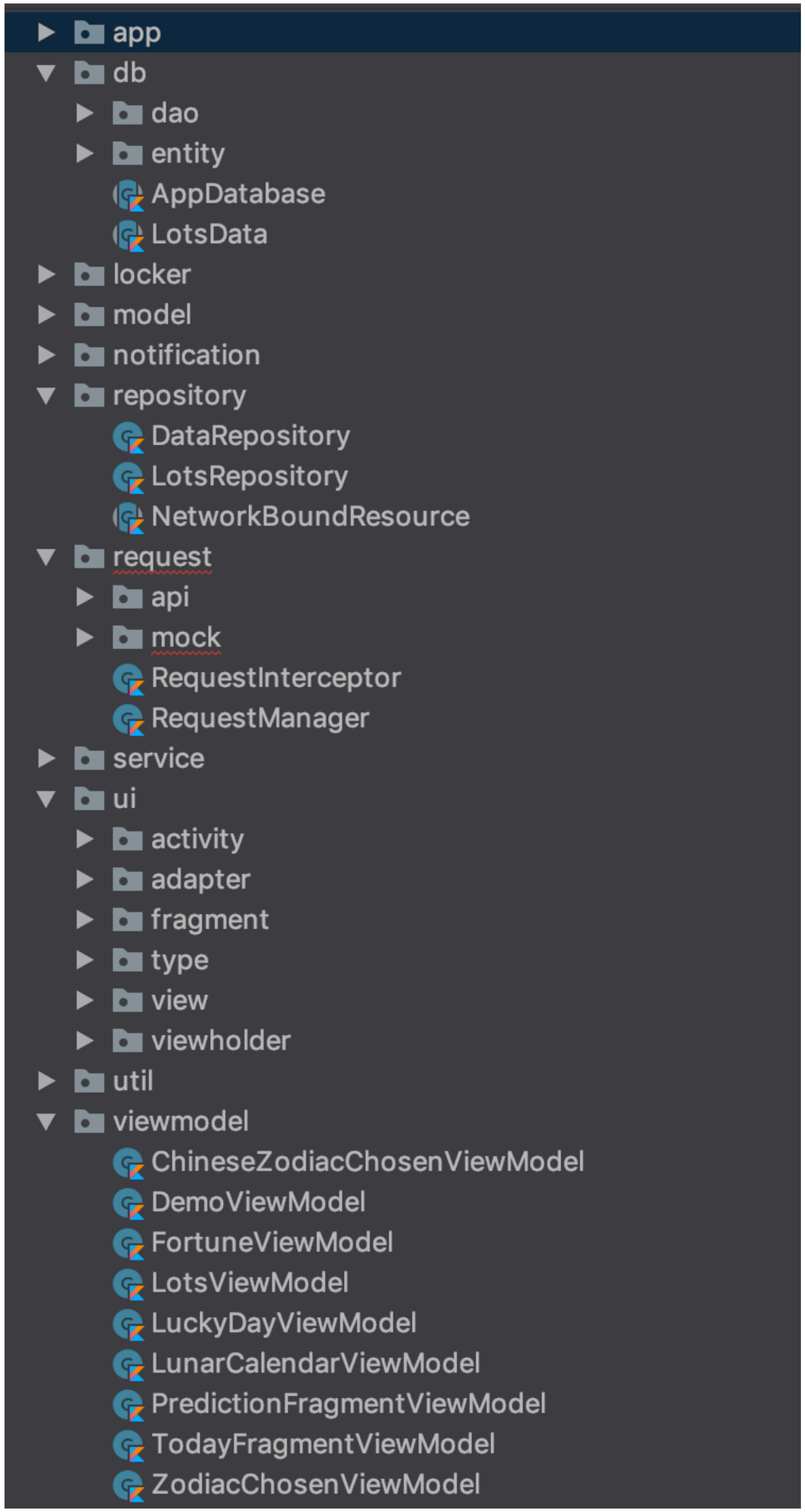
To-do App: TasksViewModel

```
fun loadTasks(forceUpdate: Boolean) {

    _dataLoading.value = true

    wrapEspressoIdlingResource {
        viewModelScope.launch { this: CoroutineScope
            val tasksResult :Result<List<Task>> = tasksRepository.getTasks(forceUpdate)

            if (tasksResult is Success) {
                val tasks :List<Task> = tasksResult.data

                val tasksToShow = ArrayList<Task>()
                // We filter the tasks based on the requestType
                for (task :Task in tasks) {
                    when (_currentFiltering) {
                        TasksFilterType.ALL_TASKS -> tasksToShow.add(task)
                        TasksFilterType.ACTIVE_TASKS -> if (task.isActive) {
                            tasksToShow.add(task)
                        }
                        TasksFilterType.COMPLETED_TASKS -> if (task.isCompleted) {
                            tasksToShow.add(task)
                        }
                    }
                }
                isDataLoadingError.value = false
                _items.value = ArrayList(tasksToShow)
            } else {
                isDataLoadingError.value = false
                _items.value = emptyList()
                showSnackbarMessage("Error while loading tasks")
            }

            _dataLoading.value = false
        }
    }
}

fun refresh() {
    loadTasks( forceUpdate: true)
}
```

我们的实践

- 目前我们使用MVVM开发并上线的应用有
  - Dragon_K
  - Panda_K
  - Number_A
  - Seiya_A
- 印象最深的一点
  - 相比MVC和MVP，写的代码更少，开发更快

Dragon文件结构


app
db
dao
entity
AppDatabase
LotsData
locker
model
notification
repository
DataRepository
LotsRepository
NetworkBoundResource
request
api
mock
RequestInterceptor
RequestManager
service
ui
activity
adapter
fragment
type
view
viewholder
util
viewmodel
ChineseZodiacChosenViewModel
DemoViewModel
FortuneViewModel
LotsViewModel
LuckyDayViewModel
LunarCalendarViewModel
PredictionFragmentViewModel
TodayFragmentViewModel
ZodiacChosenViewModel

MVVM有缺点吗?

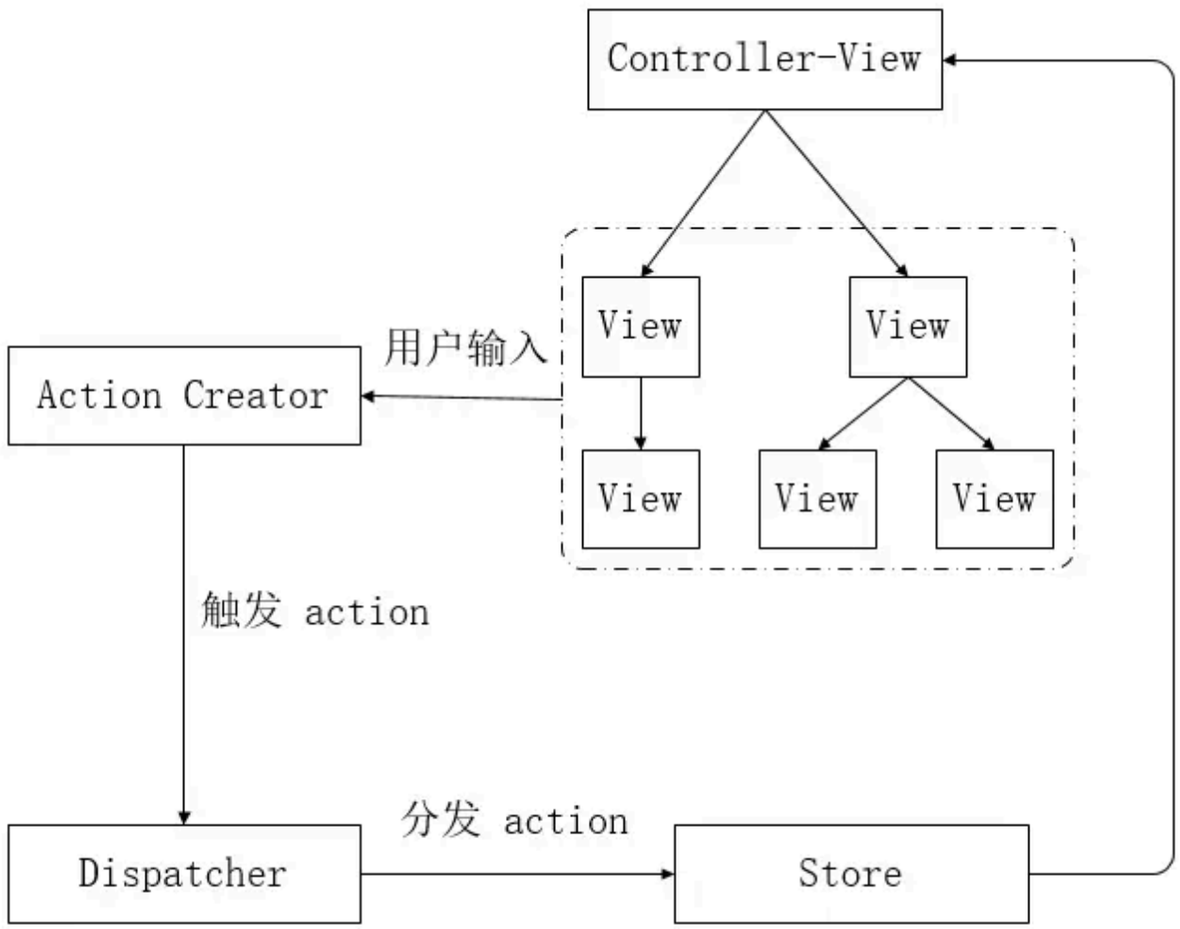
- 引入了一个更复杂的框架，有一定的学习成本
- 数据绑定需要花费更多的内存，性能有一定影响
  - 基于数据驱动的Data binding，在比较大的模块时，Model层数据可能会比较大。而且在释放的时候不能像其他模式一样，不同View的数据无法根据各自View的生命周期来进行释放，而是在整个页面销毁时统一释放，这样就会占用更大内存
- Data binding带来的问题：
  - View层UI显示是在xml文件中实现的，当出现问题时，无法确定是Model的问题还是xml的问题，而且xml中也不方便调试
  - 我们在项目中经常用到View的复用。而在databinding中 View在布局文件中可以绑定不同的model，在复用时除了要考虑View的复用之外，还需要考虑model的问题
  - 对于移动开发者而言，在布局文件中写对应的绑定数据和事件的逻辑会是一个全新的体验

MVX总结

- 不管是用哪种架构设计，只要运用得当，都可以达到想要的结果。
- 几点建议
  - 如果项目非常简单，没什么复杂性，未来改动也不大的话，那就不倾向于用设计模式或者架构方法，只需要将每个模块封装好，方便调用即可，不要为了使用设计模式或架构方法而使用。
  - 对于偏向展示型的App，绝大多数业务逻辑都在后端，app主要功能就是展示数据，交互等，建议使用mvvm。
  - 对于工具类或者需要特别多业务逻辑的App，MVP或者MVVM都可。

MVX之外的应用架构

- Flux by Facebook: 数据和逻辑永远单向流动，相邻的部分不会发生数据的双向流动。



问题讨论

- MVP中，增加了Presenter，是M-V-P之间是一种松耦合关系，是怎么做到的?
- ViewModel可以持有View或者Context的引用吗?
- MVVM中，如果因为Activity长期处于后台而被销毁了，那么重建的时候，ViewModel还是之前的实例吗?
- 除了上面讲到的，你还能想到哪些架构模式?

如何选择合适的架构?

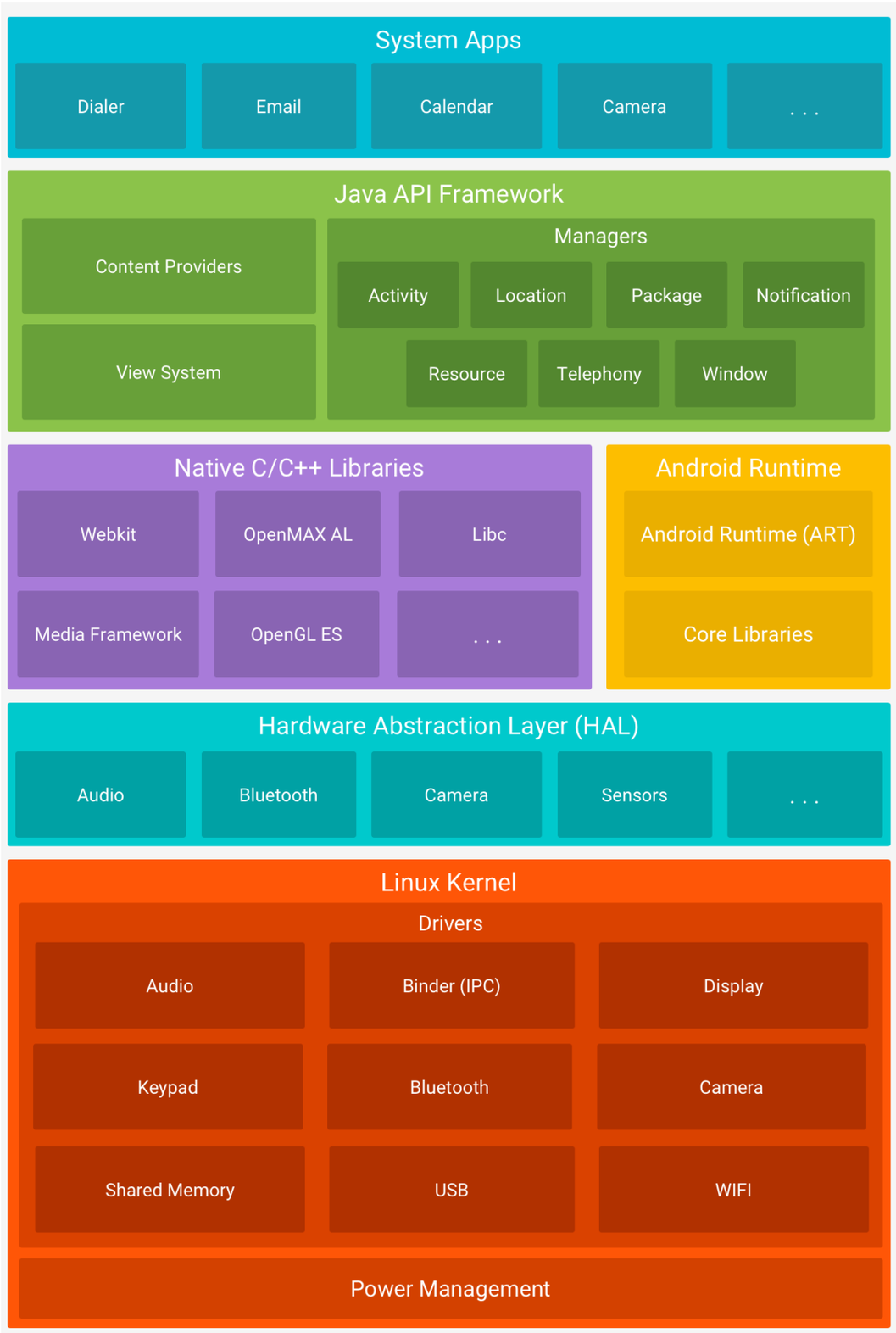
如何选择一个架构

- 什么时候才需要一个架构?
  - 新开一个项目或者需要增加一个大功能（记住 **SIZE** does matter）
  - 现有的架构遇到了瓶颈，考虑重构
- 最重要的两点
  - **封装隐藏**
  - **减小复杂度**

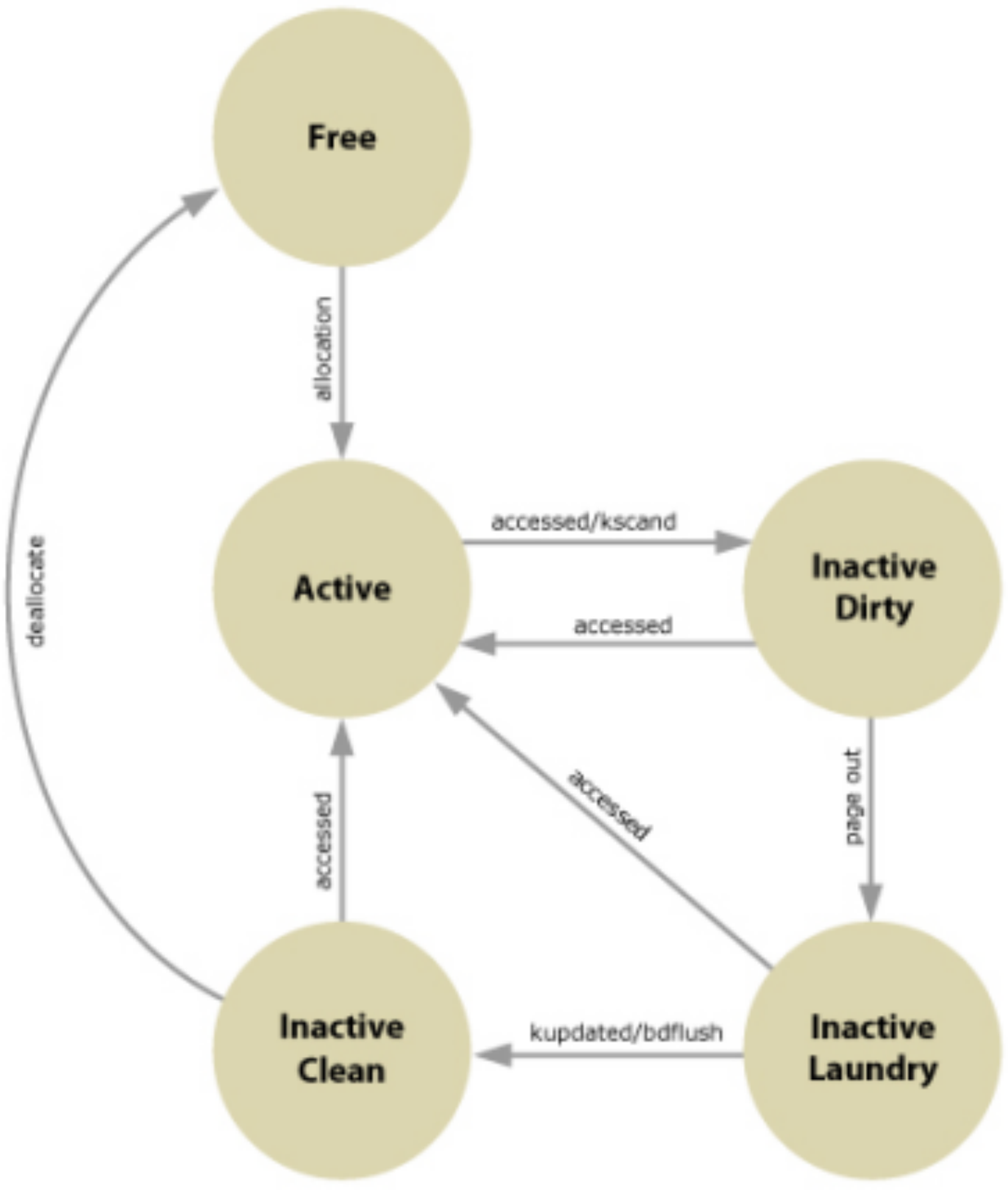
减小复杂度

- 一切的目的，就是在维持人可以思考的复杂度：
  - 1.解决问题点、2.选择工程方法、3.人员分工、4.考虑未来扩展
- 消减复杂度
  - 模块化（Modulization）
  - 状态机（State Machine）
  - 程序导向（Procedure）
  - 面向对象（Object Oriented）
- 善用设计模式 / 模型，拉高思维层级，减少复杂度

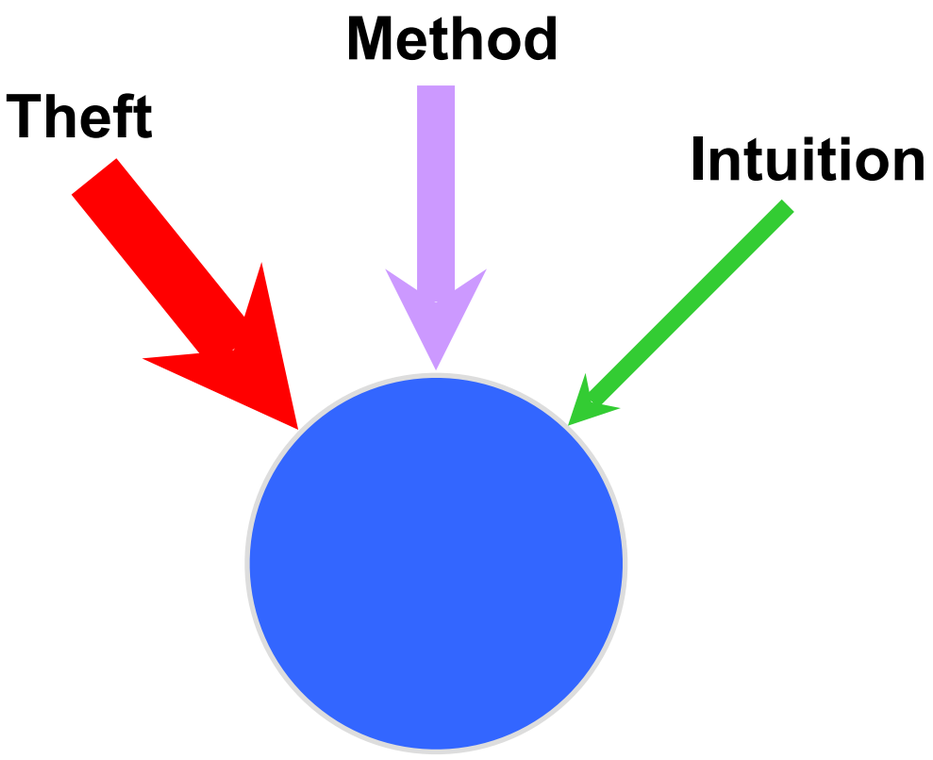
Best Practice - 1



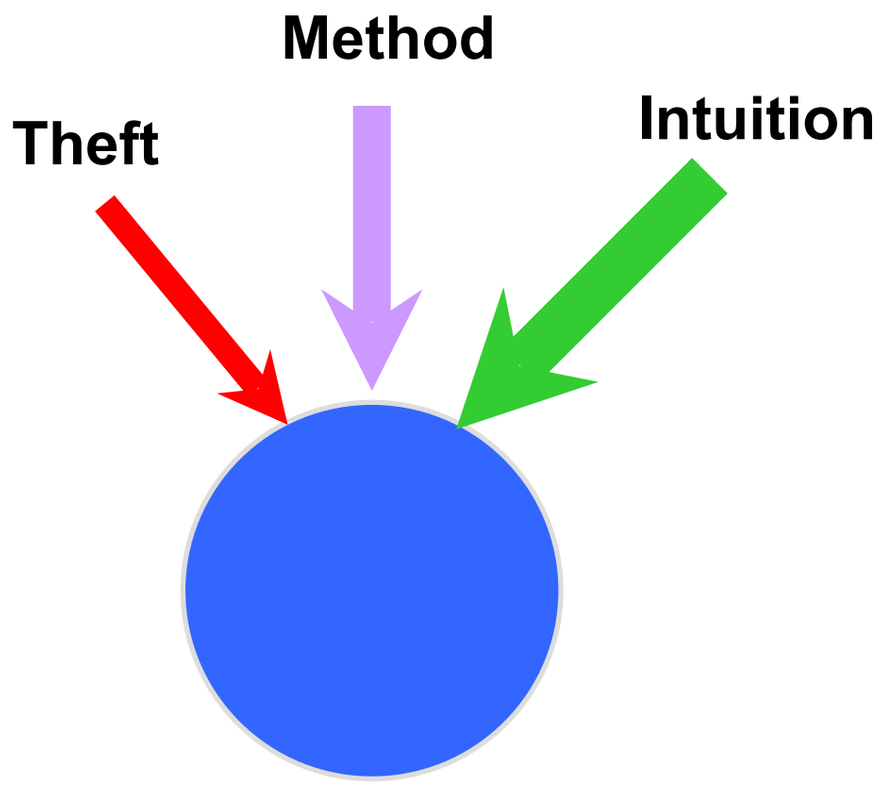
Best Practice - 2



架构从何而来

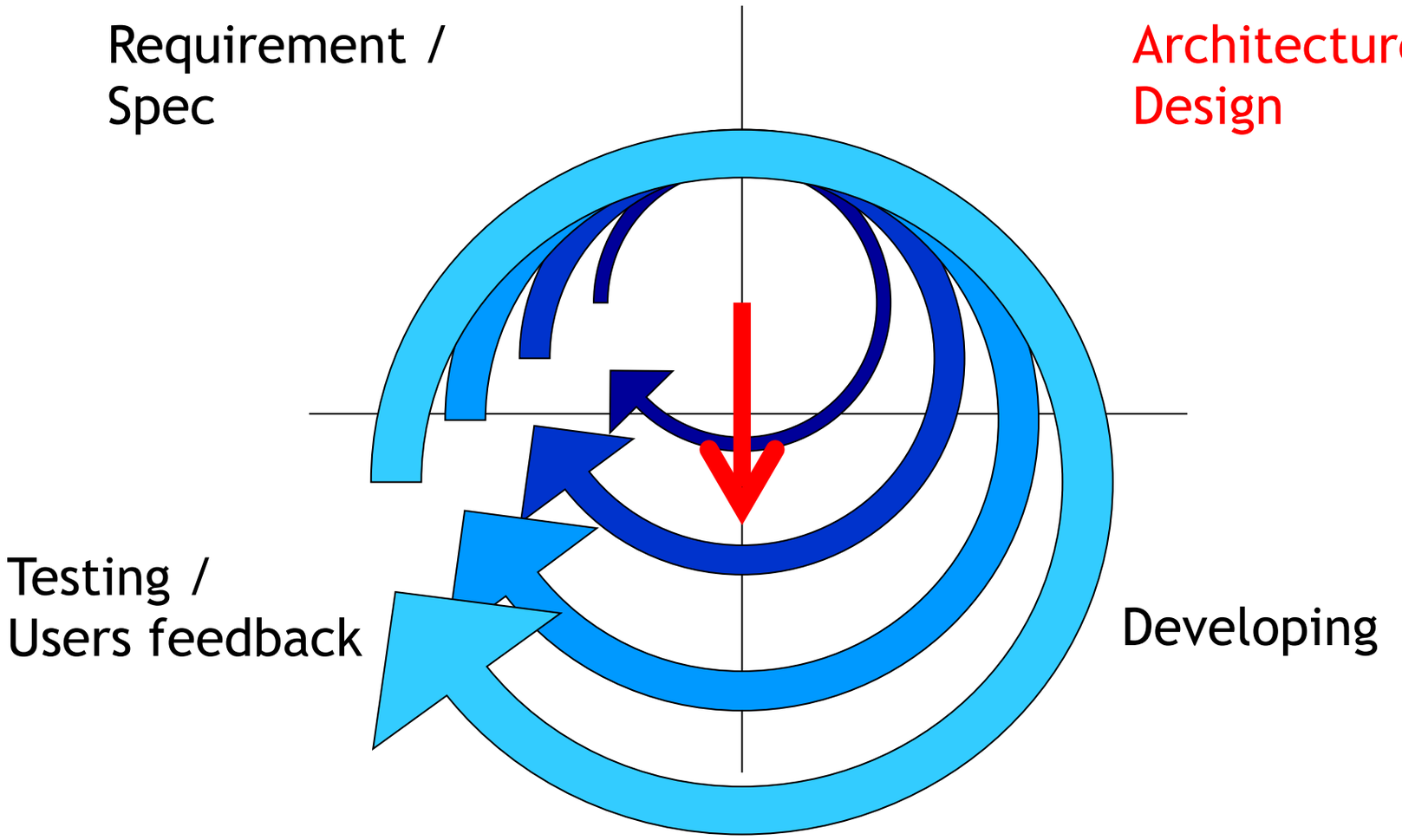


Classical system



Unprecedented system

Software develop is a loop



记住这几点

- Do not over think and over design
  - 优先考虑自己最熟悉、最拿手的设计和架构
- If you design it, you should be able to code it.
  - 学习这些模式不是为了生搬硬套，而是运用所学的知识点在需要的时候灵活运用
- Great software is not built, it is grown.
  - 从最简单的可行性的事情做起
  - 设计最小的可行系统，快速发布，快速迭代
  - 持续重构

问题讨论

- 像MVX，为什么这些流行的架构大都是分成3层或者3部分的？为什么不是4层或5层？
- 人脑无法思考过于复杂的东西，人脑有其思考的上限
- 往往天才才能同时思考4个东西
  - 为什么是4，人容易理解三维空间+时间
- 因为大部分人都不是天才，所以我们把这个数字定义为3

Albert Einstein

- It's not that I'm so smart, it's just that I stay with problems longer.
- 最贴近正解的答案，在思索这个问题最久的人身上。

Thank you